Purpose of keyword `typename`

Keyword `typename` introduces a template type parameter in a function or class template definition. The use of keyword `typename` to announce template parameters is demonstrated in the following function template definition:

```
template <typename T>
T const& Max(T const& lhs, T const& rhs) {
  return rhs < lhs ? lhs : rhs;
}
```

Keyword `typename` was introduced relatively late in the evolution of the C++98 standard. Prior to that, keyword `class` was the only way to introduce a template type parameter. To enable legacy code to compile, using keyword `class` remains a valid way to introduce template parameters:

```
template <class T>
T const& Max(T const& lhs, T const& rhs) {
  return rhs < lhs ? lhs : rhs;
}
```

From the compiler's perspective, there is no difference in the two definitions of function template `Max`. Using the keyword `class` to introduce template parameter `T` is a bit misleading since function template `Max` can be instantiated for both `class` and non-`class` types:

```
int x {10}, y {20};
std::cout << Max(x, y) << std::endl; // T is int
std::string fname {"Clint"}, lname {"Eastwood"};
std::cout << Max(fname, lname) << std::endl; // T is std::string
```

In contrast, the keyword `typename` provides a more succinct and clear mnemonic for a generic type `T` that will be replaced during template argument deduction with a valid type. Therefore, use of keyword `typename` is recommended for introducing template type parameters.

In certain situations, keyword `typename` must be used by programmers whenever their source code contains a name representing a type but that depends on a template parameter. Before introducing a specific example, let's recall that each standard container class in the standard library defines several aliases such as `size_type`, `value_type`, `iterator`, and so on to support generic programming. Programmers can use these aliases without consideration for the particular container used nor for the particular type aggregated by the container. For example, to iterate over the elements of the container, the iterator type of the container is used, which is declared as type `iterator` for read/write access and as `const_iterator` for read-only access:

```
class stlcontainer {
public:
  using iterator       = ...; // for read/write access
  using const_iterator = ...; // for read-only access
  ...
};
```

Now, consider a function template to print elements of standard library containers:

```
1  template <typename T>
2  void printcont(T const& cont, std::string const& prefix,
3                 std::string const& postfix="\n",
4                 std::ostream&      os=std::cout) {
5    os << prefix;
6    for (T::value_type const& elem : cont) {os << elem << ' '; }
7    os << postfix;
8  }
```

The `g++ -std=c++17` compiler will flag line 6 as an error because it assumes that `value_type` is a nontype member (such as a `static` data member or an enumeration constant) of class `T` that produces an expression. As a result, the definition of `elem` in `T::value_type const& elem` is considered to be syntactically incorrect. Adding keyword `template` clarifies for the compiler that `value_type` is a type defined within class `T`:

```
1  template <typename T>
2  void printcont(T const& cont, std::string const& prefix,
3                 std::string const& postfix="\n",
4                 std::ostream&      os=std::cout) {
5    os << prefix;
6    for (typename T::value_type const& elem : cont) { os << elem << ' '; }
7    os << postfix;
8  }
```

Type `T::value_type` is a dependent type because the type depends on template parameter `T`. Since `T` can refer to any type, the compiler cannot safely reach any conclusion about identifier `value_type`. The language definition resolves this problem by specifying that in general a dependent qualified name does not denote a type unless that name is prefixed with keyword `typename`. If it turns out, after substituting template arguments, that the name is not the name of a type, the program is invalid and the compiler should flag an error at instantiation time. Finally, note that this use of keyword `typename` differs from the use to denote template type parameters. Unlike type parameters, `typename` cannot be replaced with `class`.

Update on 08/12/2018

A proposal to make keyword `typename` optional has been passed by the Core Working Group and will be incorporated in C++20.

Update on 06/25/2019

The short note describing keyword `typename` was written circa 2012. Unlike `g++ -std=c++17`, the Microsoft compiler cleanly compiles the first version of the function template even though the compiler doesn't implement the C++20 feature described in the previous update.